Communication Closed Layers

Michael Harder

June 20, 2003

Introduction and Overview

The basic intuition of transformational design:

- Program design is guided by transformation rules.
- Derive the (i+1)-th implementation by applying a transformation rule on the i-th implementation,

$$1.\ implementation\ \rightarrow \ldots \rightarrow\ i - th\ implementation$$
$$\rightarrow\ (i+1) - th\ implementation\ \rightarrow \ldots$$

- Start derivation with the most coarse grained (simple) implementation satisfying the specification and proof its correctness.
- Applying transformation rules preserves correctness.
- Obtain a history of the design process by recording every transformation step.

For what good is transformational design in concurrent programs?

- Assume we need a distributed program S_D to perform some task t.
- S_D is likely to be hard to verify and the final assertional correctness proof does not reflect the design process of S_D.
- Suppose we have an implementation of t consisting of sequentially composed *parts (layers)*, say S_L.
- Usually S_L ought to be easier to develop than the distributed version S_D.
- We may also assume that S_L is easier to understand and easier to verify than S_D.

Thus we might ask:

Is there way to transform S_L into S_D such that both programs are equivalent and the correctness of S_L is preserved by the transformations?

Obviously this leads to the following questions:

1. *Of what syntactic structure is* S_L *supposed to be?*
2. *When do we consider two programs as "equivalent"?*
3. *What kind of transformations are done?*
4. *What are the transformation rules?*

Clarify the first and second question by

- defining the syntax of the programming language to employ ,
- defining equivalence of two programs. In order to do so we have to define proper semantics for our programming language.

Clarify the third and fourth question by

- introducing the *communication-closed-layers* paradigm ,
- introducing the so-called *CCL Laws*.

Syntax and Semantics

- Choose a guarded command language with
  - non-deterministic conditional command,
  - non-deterministic multi-test loop.
- Additionally:
  - every guarded command (called *action*) has a unique label,
  - parallel composition, denoted by "$\|$" and
  - **send** and **receive** actions for asynchronous communication.
- Syntax (and semantics) are closely related to transition systems. For a statement S we define $T[\![S]\!]$ the be a corresponding transition system.

Syntax

Actions	act	$::=$	$\langle b \rightarrow \bar{x} := \bar{e} \rangle \mid$
			$\mathbf{send}(c, e) \mid \mathbf{receive}(c, x)$
Programs	S	$::=$	$a : act \mid S_1 ; S_2 \mid [S_1 \| S_2] \mid$
			$\mathbf{if}\ []_{i=1}^{n} S_i\ \mathbf{fi} \mid \mathbf{do}\ S_B [] S_E ; \mathbf{exit}\ \mathbf{od}$
Closed programs	Sys	$::=$	$\langle S \rangle$

$$\mathbf{send}(c, e) \stackrel{\text{def}}{=} \langle \neg c.full \rightarrow c.full, c.buf := true, e \rangle$$

$$\mathbf{receive}(c, x) \stackrel{\text{def}}{=} \langle c.full \rightarrow c.full, x := false, c.buf \rangle$$

To the second question ...

- Transformations are supposed to preserve "equivalence" of two programs.
- The more stronger our requirements on two programs to be "equivalent" are the less room we have for interesting transformations, e.g.

$$P_1 = P_2 \text{ iff } Comp[\![P_1]\!] = Comp[\![P_2]\!].$$

- We will require that "equivalent" programs define the same initial-final state relation. This implies:
  Equivalent programs satisfy the same Hoare formulae for *partial* correctness.
- We do not require that "equivalent" programs have the same deadlock behavior or that they have the same divergence behavior.

Semantics

- Define semantics of a program S:
  - in a compositional way,
  - by sequences of action labeled computation steps, e.g.

$$\langle \sigma_0 \xrightarrow{a} \sigma_1 \rangle \langle \sigma_1' \xrightarrow{a'} \sigma_2' \rangle \ldots,$$

  - by the set of all possible sequences produced by S.
- Consider two programs as equivalent if they define the same sets of initial-final state pairs.

Reactive-event-sequence semantics $\mathcal{RA}$

- $\mathcal{RA}[\![a : \langle b \rightarrow \bar{x} := \bar{e}\rangle]\!] \stackrel{\text{def}}{=} \{\langle \sigma \stackrel{a}{\rightarrow} \sigma'\rangle \mid \sigma \models b \wedge \sigma' = \mathcal{A_I}[\![\bar{x} := \bar{e}]\!]\sigma\}$
- $\mathcal{RA}[\![S_1; S_2]\!] \stackrel{\text{def}}{=} \mathcal{RA}[\![S_1]\!]^\frown \mathcal{RA}[\![S_2]\!]$
- $\mathcal{RA}[\![\mathbf{if}\ []_{i=1}^n S_i\ \mathbf{fi}]\!] \stackrel{\text{def}}{=} \bigcup_{i=1}^n \mathcal{RA}[\![S_i]\!]$
- Let $R^{(0)} \stackrel{\text{def}}{=} \mathcal{RA}[\![S_E]\!]$, and let $R^{(i+1)} \stackrel{\text{def}}{=} \mathcal{RA}[\![S_B]\!]^\frown R^{(i)}$, for $i \geq 0$. Then:
$$\mathcal{RA}[\![\mathbf{do}\ S_B [] S_E; \mathbf{exit}\ \mathbf{od}]\!] \stackrel{\text{def}}{=} \bigcup \{R^{(i)} \mid i \geq 0\}.$$
- $\mathcal{RA}[\![[S_1 \| S_2]]\!] \stackrel{\text{def}}{=} \mathcal{RA}[\![S_1]\!]\ \tilde{\|}\ \mathcal{RA}[\![S_2]\!]$
- $\mathcal{RA}[\![\langle S\rangle]\!] \stackrel{\text{def}}{=} \{\Theta \in \mathcal{RA}[\![S]\!] \mid \Theta \text{ is connected}\}$

Limitations:

- Semantics of divergent computations are not defined.
- Sequences must be finite (the set of all sequences produced by S may be infinite).
- Deadlock behavior is not observable.

$\rightarrow$ *Partial correctness only.*

Initial-final-state semantics

- $\mathcal{O}_{cl}[\![S]\!] \stackrel{\text{def}}{=} \mathcal{O}\,[\![T[\![S]\!]]\!]$.
- $IO : \mathcal{RA} \rightarrow \Sigma^2$, constructs the initial-final state pair from a (finite) reactive sequence.
- $\mathcal{O}_{cl}[\![S]\!] \subseteq \{IO(\eta) \mid \eta \in \mathcal{RA}[\![\langle S \rangle]\!]\}$.

io-equivalence

- $S_1 \stackrel{IO}{=} S_2$ iff $\mathcal{O}_{cl}[\![S_1]\!] = \mathcal{O}_{cl}[\![S_2]\!]$.

To the third question ...

Remember our introducing example of two programs S_L and S_D implementing the same task t:

- We seek to increase/reduce concurrency of programs by applying transformation rules on them

$$S_L \underset{\textit{decrease concurrency}}{\overset{\textit{increase concurrency}}{\rightleftarrows}} S_D \ .$$

- Transformed programs are supposed to have the same initial/final-state behavior, i.e.

$$S_L \overset{IO}{=} S_D.$$

To the third question ...

- Consider a *layer* as a top-level parallel composition of program fragments, e.g. $[S_{0,0} \| S_{0,1} \| S_{0,2}]$
- Programs consist of sequentially composed *layers*, e.g. $[S_{0,0} \| S_{0,1} \| S_{0,2}]; ...; [S_{n,0} \| S_{n,1} \| S_{n,2}]$
- Increase/reduce concurrency by merging/sequentializing the layers of a program, e.g.

$$\begin{array}{l} [S_{0,0} \quad \| \quad S_{0,1} \quad \| \quad S_{0,2}]\,; \\ [S_{1,0} \quad \| \quad S_{1,1} \quad \| \quad S_{1,2}]\,; \\ [S_{2,0} \quad \| \quad S_{2,1} \quad \| \quad S_{2,2}] \end{array} \quad \leftrightsquigarrow \quad \left[\begin{array}{c} S_{0,0}; \\ S_{1,0}; \\ S_{2,0}; \end{array} \middle\| \begin{array}{c} S_{0,1}; \\ S_{1,1}; \\ S_{2,1}; \end{array} \middle\| \begin{array}{c} S_{0,2} \\ S_{1,2} \\ S_{2,2} \end{array} \right].$$

The Communication-Closed-Layers Laws

Introducing four CCL Laws (CCL 1 - CCL 4):

- CCL 1 and CCL 2 for programs using shared variables.
- CCL 3 and CCL 4 for *communication based* programs, i.e. no shared variables are used except those associated with the communication buffers.

Additionally we will inquire how to transform loops:

- Loop Distribution Theorem for while-loops.

CCL Laws for Shared Variables

Syntactically commuting and conflicting

Two say that actions $a_1 \equiv \langle b_1 \rightarrow \bar{x}_1 := \bar{e}_1 \rangle$ and $a_2 \equiv \langle b_2 \rightarrow \bar{x}_2 := \bar{e}_2 \rangle$ are *syntactically commuting* if:

(i) $write(a_1) \cap read(a_2) = \emptyset$.

(ii) $write(a_2) \cap read(a_1) = \emptyset$.

(iii) $write(a_1) \cap write(a_2) = \emptyset$.

Actions which do not syntactically commute are said to be $in\ conflict$.

We introduce the following notations:

- $a_1 \not\frown a_2$, if a_1 and a_2 are syntactically commuting
- $a_1 \frown a_2$, if a_1 and a_2 are in conflict

For program fragments S_1 and S_2 we define

- $S_1 \not\frown S_2$

by requiring that for all a_1 occurring in S_1 and all a_2 occurring in S_2 we have that $a_1 \not\frown a_2$.

Concurrent actions

Two actions a and a' occurring in S are called *concurrent* actions if there are two different parallel components S_1 and S_2, such that a occurs in S_1 and a' occurs in S_2.

Example

- $S \equiv S_1; [S_2 \| (S_3; [S_4 \| S_5]; S_6)]$.

Parallel components of S:

- S_2, S_4, S_5 and $(S_3; [S_4 \| S_5]; S_6)$.

Let a_i be an action occurring in S_i for $i \in \{1, 2, 3, 4, 5, 6\}$, then

- $(a_2,\, a_3)$ are concurrent
- $(a_3,\, a_4)$ are not concurrent

Commuting Actions Lemma

Consider

- closed program $\langle S \rangle$,
- a_i and a_{i+1} concurrent actions occurring in $\langle S \rangle$ s.t. $a_i \not\longrightarrow a_{i+1}$.

Let $\eta \in \mathcal{RA}[\![\langle S \rangle]\!]$ of the form:

$$\langle \sigma_0 \stackrel{a_0}{\rightarrow} \sigma_1 \rangle \ldots \langle \sigma_{i-1} \stackrel{a_{i-1}}{\rightarrow} \sigma_i \rangle \langle \sigma_i \stackrel{a_i}{\rightarrow} \sigma_{i+1} \rangle \langle \sigma_{i+1} \stackrel{a_{i+1}}{\rightarrow} \sigma_{i+2} \rangle \langle \sigma_{i+1} \stackrel{a_{i+2}}{\rightarrow} \sigma_{i+3} \rangle \ldots$$

Let η' be *defined as* η *with* a_i *and* a_{i+1} *exchanged*, i.e., of the form:

$$\langle \sigma_0 \stackrel{a_0}{\rightarrow} \sigma_1 \rangle \ldots \langle \sigma_{i-1} \stackrel{a_{i-1}}{\rightarrow} \sigma_i \rangle \langle \sigma_i \stackrel{a_{i+1}}{\rightarrow} \sigma'_{i+1} \rangle \langle \sigma'_{i+1} \stackrel{a_i}{\rightarrow} \sigma_{i+2} \rangle \langle \sigma_{i+1} \stackrel{a_{i+2}}{\rightarrow} \sigma_{i+3} \rangle \ldots$$

Then $\eta' \in \mathcal{RA}[\![\langle S \rangle]\!]$.

Proof: Suppose $a_i \equiv \langle b_i \rightarrow \bar{x}_i := \bar{e}_i \rangle$ and $a_{i+1} \equiv \langle b_{i+1} \rightarrow \bar{x}_{i+1} := \bar{e}_{i+1} \rangle$.
To prove $\eta' \in \mathcal{RA}[\![\langle S \rangle]\!]$ we have to show

(i) $[\![b_{i+1}]\!](\sigma_i) = tt$,

(ii) $[\![b_i]\!](\sigma'_{i+1}) = tt$, where $\sigma'_{i+1} = [\![\bar{x}_{i+1} := \bar{e}_{i+1}]\!](\sigma_i)$,

(iii) $[\![\bar{x}_i := \bar{e}_i]\!](\sigma'_{i+1}) = [\![\bar{x}_{i+1} := \bar{e}_{i+1}]\!](\sigma_{i+1})$,

Since $\eta \in \mathcal{RA}[\![\langle S \rangle]\!]$ we have that

(a) $[\![b_i]\!](\sigma_i) = tt$

(b) $[\![b_{i+1}]\!](\sigma_{i+1}) = tt$

(c) $\sigma_{i+1} = [\![\bar{x}_i := \bar{e}_i]\!](\sigma_i)$ and $\sigma_{i+2} = [\![\bar{x}_{i+1} := \bar{e}_{i+1}]\!](\sigma_{i+1})$.

By $a_i \not\longrightarrow a_{i+1}$ we have

($\star$) $write(a_i) \cap write(a_{i+1}) = \emptyset$.

It follows:

$$\begin{aligned}
[\![\bar{x}_i := \bar{e}_i]\!](\sigma'_{i+1}) &= [\![\bar{x}_i := \bar{e}_i]\!]([\![\bar{x}_{i+1} := \bar{e}_{i+1}]\!](\sigma_i)) \\
&\overset{(\star)}{=} [\![\bar{x}_{i+1} := \bar{e}_{i+1}]\!]([\![\bar{x}_i := \bar{e}_i]\!](\sigma_i)) \\
&\overset{(c)}{=} [\![\bar{x}_{i+1} := \bar{e}_{i+1}]\!](\sigma_{i+1}).
\end{aligned}$$

Thus, point (iii) is valid. Now consider (i) and (ii): By definition

- $[\![b_{i+1}]\!](\sigma_{i+1}) = [\![b_{i+1}]\!]([\![\bar{x}_i := \bar{e}_i]\!](\sigma_i)) \overset{(b)}{=} tt$,
- $[\![b_i]\!](\sigma'_{i+1}) = [\![b_i]\!]([\![\bar{x}_{i+1} := \bar{e}_{i+1}]\!](\sigma_i))$.

From $[\![b_{i+1}]\!](\sigma_{i+1}) = tt$ and $write(a_i) \cap read(a_{i+1}) = \emptyset$ it follows

$$[\![b_{i+1}]\!](\sigma_i) = tt.$$

Since $[\![b_i]\!](\sigma_i) = tt$ and $write(a_{i+1}) \cap read(a_i) = \emptyset$ we conclude

$$[\![b_i]\!](\sigma_i) = tt.$$

CCL 1 (Independent program fragments)

Let S_L and S_D be programs defined as follows:

$$S_L \stackrel{\text{def}}{=} \begin{array}{c} \left[\; S_{0,0} \;\|\; \ldots \;\|\; S_{0,m} \;\right] \\ ; \\ \vdots \quad\quad \vdots \quad\quad \vdots \\ ; \\ \left[\; S_{n,0} \;\|\; \ldots \;\|\; S_{n,m} \;\right] \end{array} \quad \text{and} \quad S_D \stackrel{\text{def}}{=} \left[\begin{array}{c} S_{0,0} \\ ; \\ \vdots \\ ; \\ S_{n,0} \end{array} \middle\| \begin{array}{c} \ldots \\ \ldots \\ \vdots \\ \ldots \\ \ldots \end{array} \middle\| \begin{array}{c} S_{0,m} \\ ; \\ \vdots \\ ; \\ S_{n,m} \end{array} \right].$$

Assume that $S_{i,j} \not\longrightarrow S_{i',j'}$ for $i \neq i'$ and $j \neq j'$, then $S_L \stackrel{IO}{=} S_D$.

Proof: Follows from CCL 2.

CCL 2 (Conflict-based ordering)

Let S_L and S_D be programs defined as follows:

$$S_L \stackrel{\text{def}}{=} \begin{array}{c} [\, S_{0,0} \parallel \ldots \parallel S_{0,m} \,] \\ ; \\ \vdots \quad\quad \vdots \quad\quad \vdots \\ ; \\ [\, S_{n,0} \parallel \ldots \parallel S_{n,m} \,] \end{array} \quad \text{and} \quad S_D \stackrel{\text{def}}{=} \left[\begin{array}{c} S_{0,0} \\ ; \\ \vdots \\ ; \\ S_{n,0} \end{array} \middle\| \begin{array}{c} \ldots \\ \ldots \\ \vdots \\ \ldots \\ \ldots \end{array} \middle\| \begin{array}{c} S_{0,m} \\ ; \\ \vdots \\ ; \\ S_{n,m} \end{array} \right].$$

Assume that $\models \langle S_D \rangle$ **psat** $(S_{i,j} \stackrel{C}{\twoheadrightarrow} S_{i',j'})$ holds for all $i < i'$ and $j \neq j'$. Then $S_L \stackrel{IO}{=} S_D$.

Proof: We show

(i) $\forall \eta_L \in \mathcal{RA}[\![\langle S_L \rangle]\!].\exists \eta_D \in \mathcal{RA}[\![\langle S_D \rangle]\!]$ s.t. $IO(\eta_L) = IO(\eta_D)$

(ii) $\forall \eta_D \in \mathcal{RA}[\![\langle S_D \rangle]\!].\exists \eta_L \in \mathcal{RA}[\![\langle S_L \rangle]\!]$ s.t. $IO(\eta_D) = IO(\eta_L)$

Since any sequence produced by $\langle S_L \rangle$ is also produced by $\langle S_D \rangle$, i.e.

$$\mathcal{RA}[\![\langle S_L \rangle]\!] \subseteq \mathcal{RA}[\![\langle S_D \rangle]\!],$$

it remains to prove point (ii). Let $\eta_D \in \mathcal{RA}[\![\langle S_D \rangle]\!]$.
Consider the case $n = 1$:

$$S_L = \begin{array}{ccccc} [S_{0,0} & \| & \cdots & \| & S_{0,m}] \\ & & ; & & \\ [S_{1,0} & \| & \cdots & \| & S_{1,m}] \end{array} \quad \text{and } S_D = \left[\begin{array}{c} S_{0,0} \\ ; \\ S_{1,0} \end{array} \middle\| \begin{array}{c} \cdots \\ \cdots \\ \cdots \end{array} \middle\| \begin{array}{c} S_{0,m} \\ ; \\ S_{1,m} \end{array} \right].$$

There might be occurrences of events a_0 and a_1 in $\eta_{_D}$, where a_1 labels an action in $S_{1,j}$ and a_0 labels an action in $S_{0,j'}$, s.t. the a_1-event precedes the a_0-event (note: this is not possible in $\eta_{_L}$). Let k denote the number of such event pairs (a_1, a_0) occurring in $\eta_{_D}$.

We show by induction on k:

$$(\star) \quad \forall k. \exists \eta_{_L} \in \mathcal{RA}[\![\langle S_L \rangle]\!] \text{ s.t. } IO(\eta_{_L}) = IO(\eta_{_D}).$$

- **Basis case:**
  Let $k = 0$. Then we have that $\eta_{_D} \in \langle S_L \rangle$.

- **Induction step:**
  Assume that $\eta_{_D}$ has $k+1$ pairs of events (a_1, a_0) as indicated.
  We may assume that there exists a pair (a_1, a_0) in $\eta_{_D}$ s.t. a_0 immediately follows a_1, thus $\eta_{_D}$ is of the following form:

$\eta_D = \theta\langle\sigma \xrightarrow{a_1} \sigma'\rangle\langle\sigma' \xrightarrow{a_0} \sigma''\rangle\theta'$, for some sequences θ, θ' and states σ, σ', σ''.

- From $\models \langle S_D\rangle$ **psat** $(S_{0,j} \overset{C}{\twoheadrightarrow} S_{1,j'})$ for all $j \neq j'$ it follows

$$a_1 \not\longrightarrow a_0.$$

- Hence, by Commuting Actions Lemma we have that

$$\theta\langle\sigma \xrightarrow{a_0} \tau'\rangle\langle\tau' \xrightarrow{a_1} \sigma''\rangle\theta' \in \mathcal{RA}[\![\langle S_D\rangle]\!].$$

- Applying the ind. hypothesis on $\eta\langle\sigma \xrightarrow{a_0} \tau'\rangle\langle\tau' \xrightarrow{a_1} \sigma''\rangle\eta'$ we obtain

$$\exists\eta_L \in \mathcal{RA}[\![\langle S_L\rangle]\!] \text{ s.t. } IO(\eta_L) = IO\left(\theta\langle\sigma \xrightarrow{a_0} \tau'\rangle\langle\tau' \xrightarrow{a_1} \sigma''\rangle\theta'\right).$$

The case $n > 1$ can be shown analogously by defining a k for every event pair $(a_i, a_{i'})$ with $i > i'$.

Example

Consider the following program S given by

$$S_D \stackrel{\text{def}}{=} \left[\begin{array}{l} a_1 : z := 2; \\ \\ P_1 : P(s); \\ \\ a_2 : w := 1; \\ \\ P_2 : P(s); \\ \\ a_3 : z := x + 1; \end{array} \middle\| \begin{array}{l} a_1' : x := 2; \\ \\ V_1 : V(s); \\ \\ a_2' : v := 1; \\ \\ V_2 : V(s); \\ \\ a_3' : v := w + 1; \end{array} \right].$$

We claim that $\models \{s\}\langle\, S \,\rangle\{z = 3 \wedge v = 2\}$.

CCL Laws for Shared Variables

Applying the proof method of Owickie & Gries we would have to verify:

- max. 60 verification conditions to proof interference freedom

Instead we transform S into a layered version: Let

$$S_{0,0} \stackrel{\text{def}}{=} \begin{cases} a_1 : z := 2; \\ \\ P_1 : P(s); \\ \\ a_2 : w := 1; \end{cases} , \qquad S_{0,1} \stackrel{\text{def}}{=} \begin{cases} a_1' : x := 2; \\ \\ V_1 : V(s); \\ \\ a_2' : v := 1; \end{cases}$$

$$S_{1,0} \stackrel{\text{def}}{=} \begin{cases} P_2 : P(s); \\ \\ a_3 : z := x + 1; \end{cases} , \quad S_{1,1} \stackrel{\text{def}}{=} \begin{cases} V_2 : V(s); \\ \\ a_3' : v := w + 1; \end{cases} .$$

Then

$$S_D = \left[\begin{array}{c|c} S_{0,0}; & S_{0,1}; \\ S_{1,0}; & S_{1,1}; \end{array} \right].$$

Let $\mathrm{S_L}$ be the layered version of $\mathrm{S_D}$, i.e.

$$S_L \stackrel{\text{def}}{=} \begin{cases} [S_{0,0} \quad \| \quad S_{0,1}]\,; \\ [S_{1,0} \quad \| \quad S_{1,1}] \end{cases}.$$

We claim that $S_D \stackrel{IO}{=} S_L$ by CCL 2.

To proof correctness of S_L using O&G method we have to verify:

- max. 36 verification conditions to proof interference freedom

Are the requirements of CCL 2 satisfied?

We have to prove $\models \langle S_D \rangle$ **psat** $(S_{i,j} \stackrel{C}{\twoheadrightarrow} S_{i',j'})$ for all $i < i'$ and $j \neq j'$, i.e.

(a) $\models \langle S_D \rangle$ **psat** $(P_1 \stackrel{C}{\twoheadrightarrow} V_2)$

(b) $\models \langle S_D \rangle$ **psat** $(a_2 \stackrel{C}{\twoheadrightarrow} a_3')$

(c) $\models \langle S_D \rangle$ **psat** $(a_1' \stackrel{C}{\twoheadrightarrow} a_3)$

(d) $\models \langle S_D \rangle$ **psat** $(V_1 \stackrel{C}{\twoheadrightarrow} P_2)$

Conditions (a) and (d) are valid by the ordering caused by semaphores. We can deduce the validity of (b) and (c) by the ordering caused by sequential composition.

CCL Laws for Communication-Based Programs

Communication-based

We call a programm S *communication-based* if there are no shared variables used in S except those associated with the channels. All communication is done using **send** and **receive** actions as defined earlier.

Syntactic send and receive counters

For a channel c we define the functions

- $ns_c(S)$, the number of **send** events in S along channel c,
- $nr_c(S)$, the number of **receive** events in S along channel c,

by induction on the syntactic structure of S...

- $ns_c(\mathbf{send}(c, e)) \stackrel{\text{def}}{=} 1$, and $ns_c(a) \stackrel{\text{def}}{=} 0$ for all other atomic actions a,

- $ns_c(S_1; S_2) \stackrel{\text{def}}{=} \begin{cases} ns_c(S_1) + ns_c(S_2) & : \quad ns_c(S_1) \neq \perp \wedge ns_c(S_2) \neq \perp \\ \perp & : \quad \text{otherwise} \end{cases}$,

- $ns_c(\mathbf{if}\ []_{i=1}^{n} b_i \rightarrow S_i\ \mathbf{fi}) \stackrel{\text{def}}{=} \begin{cases} nr_c(S_1) & : \quad \forall_{1 \leq i,j \leq n}(ns_c(S_i) = ns_c(S_j)) \\ \perp & : \quad \text{otherwise} \end{cases}$,

- $ns_c([S_1 \parallel S_2]) \stackrel{\text{def}}{=} \begin{cases} ns_c(S_1) + ns_c(S_2) & : \quad ns_c(S_1) \neq \perp \wedge ns_c(S_2) \neq \perp \\ \perp & : \quad \text{otherwise} \end{cases}$,

- $ns_c(\mathbf{do}\ []_{i=1}^{n} b_i \rightarrow S_i\ \mathbf{od}) \stackrel{\text{def}}{=} \begin{cases} 0 & : \quad \forall_{1 \leq i,j \leq n}(ns_c(S_i) = 0)) \\ \perp & : \quad \text{otherwise} \end{cases}$.

Analogously we define nr_c.

Syntax-based communication closedness

- Let $L \stackrel{\text{def}}{=} [S_0 \parallel \ldots \parallel S_m]$ be a layer that uses some channel c.
- Assume that S_i contains *all* the **send** actions for c and some S_j with $i \neq j$ contains *all* the **receive** actions for c.

L is called *communication closed for a channel c* if

$$\perp \neq ns_c(S_i) = nr_c(S_j) \neq \perp .$$

L is called *communication closed* if it is communication closed for all channels occurring in L.

Example

The layer

$$\left[\begin{array}{l\|l} \textbf{send}(c,e); & \textbf{receive}(c,e); \\ \textbf{receive}(d,x) & \textbf{send}(d,x) \end{array}\right]$$

is communication closed, however

$$\left[\begin{array}{l\|l} \textbf{send}(c,e); & \textbf{receive}(c,x); \\ \textbf{receive}(d,x) & \textbf{send}(c,e) \end{array}\right]$$

is *not* communication-closed.

CCL 3 (Syntax-based CCL)

Let S_L and S_D be communication-based programs defined as follows:

$$S_L \stackrel{\text{def}}{=} \begin{array}{c} \left[\; S_{0,0} \parallel \ldots \parallel S_{0,m} \;\right] \\ ; \\ \vdots \qquad \vdots \qquad \vdots \\ ; \\ \left[\; S_{n,0} \parallel \ldots \parallel S_{n,m} \;\right] \end{array} \quad \text{and} \quad S_D \stackrel{\text{def}}{=} \left[\begin{array}{c} S_{0,0} \\ ; \\ \vdots \\ ; \\ S_{n,0} \end{array} \middle\| \begin{array}{c} \ldots \\ \ldots \\ \vdots \\ \ldots \\ \ldots \end{array} \middle\| \begin{array}{c} S_{0,m} \\ ; \\ \vdots \\ ; \\ S_{n,m} \end{array} \right].$$

Assume that each layer $L_i \stackrel{\text{def}}{=} [S_{i,0} \parallel \ldots \parallel S_{i,m}]$, where $1 \leq i \leq n$, is communication closed. Then $S_L \stackrel{IO}{=} S_D$.

Proof: By CCL 2 we have to prove:

$$\models \langle S_D \rangle \;\mathbf{psat}\; (S_{i,j} \overset{C}{\twoheadrightarrow} S_{i',j'}) \text{ for all } i < i' \text{ and } j \neq j'. \qquad (\star)$$

(b) Consider the case n=1, i.e.

$$S_L = \begin{array}{c} [S_{0,0} \quad \| \quad \cdots \quad \| \quad S_{0,m}] \\ ; \\ [S_{1,0} \quad \| \quad \cdots \quad \| \quad S_{1,m}] \end{array} \quad \text{and } S_D = \left[\begin{array}{c} S_{0,0} \\ ; \\ S_{1,0} \end{array} \middle\| \begin{array}{c} \cdots \\ \cdots \\ \cdots \end{array} \middle\| \begin{array}{c} S_{0,m} \\ ; \\ S_{1,m} \end{array} \right].$$

(c) Assume $(\star)$ is not valid. Then

$$\exists \eta_D \in \mathcal{RA}[\![\langle S_D \rangle]\!].\eta_D = \theta \langle \sigma \overset{a_1}{\rightarrow} \sigma' \rangle \langle \sigma' \overset{a_0}{\rightarrow} \sigma'' \rangle \theta' \wedge a_0 \text{---} a_1,$$

where a_0 is occurring in $S_{0,j}$ and a_1 in $S_{1,j'}$ for $j \neq j'$.

(d) Let a_1 be the *first* event preceding such an a_0-event.

Assertion-Based Program Transformations

Precondition-based semantics

$$O_{cl}[\![\, \{pre\} S \,]\!] \stackrel{\text{def}}{=} \{(\sigma, \sigma') \mid \exists \eta . \sigma \models pre \wedge (\sigma, \sigma') = IO(\eta) \wedge \eta \in \mathcal{RA}[\![\langle S \rangle]\!]\}.$$

Precondition-based io-equivalence

We define precondition-based io-equivalence between two SVL^{++} programs S_1 and S_1 denoted by

$$\{p_1\}S_1 \stackrel{IO}{=} \{p_2\}S_2,$$

if $O_{cl}[\![\{p_1\}S_1]\!] = O_{cl}[\![\{p_2\}S_2]\!]$.

Augmented send/receive

$\mathbf{send}(c, e) \stackrel{\text{def}}{=} \langle \neg c.full \rightarrow c.full, cbuf, c.sent, := true, e, c.sent + 1\rangle$

$\mathbf{receive}(c, x) \stackrel{\text{def}}{=} \langle c.full \rightarrow c.full, x, c.received := false, c.buf, c.received + 1\rangle.$

Assertion-based communication closedness

Let S be a program with precondition *pre*.

- S is called *communication closed for a channel c* if

$$\models \{pre \wedge c.sent = c.received\}\langle S\rangle\{c.sent = c.received\}.$$

- A program or layer with precondition *pre* is called *communication closed* if it is communication closed for all of its channels.

CCL 4 (Assertion based)

Let S_L and S_D be communication-based programs defined as follows:

$$S_L \stackrel{\text{def}}{=} \begin{array}{c} \{p_0\} \\ \left[\; S_{0,0} \parallel \ldots \parallel S_{0,m} \;\right] \\ ; \\ \{p_1\} \\ \vdots \qquad \vdots \qquad \vdots \\ ; \\ \{p_n\} \\ \left[\; S_{n,0} \parallel \ldots \parallel S_{n,m} \;\right] \end{array} \quad \text{and} \quad S_D \stackrel{\text{def}}{=} \left[\begin{array}{c} S_{0,0} \\ ; \\ \vdots \\ ; \\ S_{n,0} \end{array} \middle\| \begin{array}{c} \ldots \\ \ldots \\ \vdots \\ \ldots \\ \ldots \end{array} \middle\| \begin{array}{c} S_{0,m} \\ ; \\ \vdots \\ ; \\ S_{n,m} \end{array} \right].$$

Assume that each layer with precondition

$$L_i \stackrel{\text{def}}{=} \{p_i\}[S_{i,0} \parallel \ldots \parallel S_{i,m}]$$

is communication closed, and that $\{p_i\}[S_{i,0} \parallel \ldots \parallel S_{i,m}]\{p_{i+1}\}$ is valid for $0 \leq i \leq n-1$. Then $S_L \stackrel{IO}{=} S_D$.

Loop Distribution

Lemma (Loop unfolding)

For all contexts $C[\cdot]$ we have

$$O_{cl}[\![C[\mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od}]]\!] = \bigcup_{j \in \mathbb{N}} O_{cl}[\![C[(b; S)^j; \neg b]]\!].$$

Loop distribution

Consider a program **while** b **do** $[S_1 \parallel S_2]$ **od**, guards b_1, b_2, and assertions p and I with the following properties:

(i) I is a loop invariant, i.e.,

$$\models \{I \wedge b\} \, \langle [S_1 \parallel S_2] \rangle \, \{I\}.$$

(ii) $\models p \rightarrow I$.

(iii) The variables of b_i are local to S_i, $i \in \{1, 2\}$, and moreover the following is valid:

$$\models I \rightarrow ((b \leftrightarrow b_1) \wedge (b \leftrightarrow b_2)).$$

(iv) $\{I\}[S_1 \parallel S_2]$ is communication closed.

Then

$$\{p\}\ \mathbf{while}\ b\ \mathbf{do}\ [S_1 \parallel S_2]\ \mathbf{od}$$

$$\stackrel{IO}{=}$$

$$\{p\}\ [\mathbf{while}\ b_1\ \mathbf{do}\ S_1\ \mathbf{od} \parallel \mathbf{while}\ b_2\ \mathbf{do}\ S_2\ \mathbf{od}]$$

and $\{p\}\ [\mathbf{while}\ b_1\ \mathbf{do}\ S_1\ \mathbf{od} \parallel \mathbf{while}\ b_2\ \mathbf{do}\ S_2\ \mathbf{od}]$ is communication closed.

Proof: By Lemma (Loop unfolding) we have that

$$O_{cl}[\![\{p\}\textbf{while } b \textbf{ do } [S_1 \| S_2] \textbf{ od}]\!] = \bigcup_{j \in \mathbb{N}} O_{cl}[\![\{p\}(b; [S_1 \| S_2])^j; \neg b]\!].$$

(a) Let $j \in \mathbb{N}$ and $\eta \in \mathcal{RA}[\![\langle (b; [S_1 \| S_2])^j; \neg b \rangle]\!]$ s. t. the initial state of η satisfies p. Let σ be an intermediate state in η where the guard b or $\neg b$ is evaluated. Then by (i) and (ii) we have that $\sigma \models I$. Property (iii) now implies that $\sigma \models b$ iff $\sigma \models b_1 \wedge b_2$ and that $\sigma \models \neg b$ iff $\sigma \models \neg b_1 \wedge \neg b_2$. We conclude

$$\{p\}(b; [S_1 \| S_2])^j; \neg b \stackrel{IO}{=} \{p\} \underbrace{((b_1 \wedge b_2); [S_1 \| S_2])^j; (\neg b_1 \wedge \neg b_2)}_{(\star)}.$$

(b) It can be shown that $(b_1 \wedge b_2) \overset{IO}{=} [b_1 \| b_2]$ and $(\neg b_1 \wedge \neg b_2) \overset{IO}{=} [\neg b_1 \| \neg b_2]$, and since $(b_1 \wedge b_2)$ and $(\neg b_1 \wedge \neg b_2)$ are not within the scope of a parallel composition operator inside $(\star)$ we can replace $(b_1 \wedge b_2)$ and $(\neg b_1 \wedge \neg b_2)$ in (a), obtaining:

$$\{p\}((b_1 \wedge b_2); [S_1 \| S_2])^j; (\neg b_1 \wedge \neg b_2) \overset{IO}{=} \{p\}([b_1 \| b_2]; [S_1 \| S_2])^j; [\neg b_1 \| \neg b_2].$$

(c) By (ii) and (iv) we may apply CCL 4 on $\{p\}[b_1 \| b_2]; [S_1 \| S_2]$, thus

$$\{p\}[b_1 \| b_2]; [S_1 \| S_2] \overset{IO}{=} \{p\}[b_1; S_2 \| b_2; S_2].$$

With respect to (b) we conclude

$$\{p\}([b_1 \| b_2]; [S_1 \| S_2])^j; [\neg b_1 \| \neg b_2] \overset{IO}{=} \{p\}([b_1; S_2 \| b_2; S_2])^j; [\neg b_1 \| \neg b_2].$$

(d) By properties (ii) and (iv) $\{p\}[b_1; S_1 \| b_2; S_2]; [b_1; S_1 \| b_2; S_2]$ satisfies the requirements of CCL 4, hence

$$\{p\}[b_1; S_1 \| b_2; S_2]; [b_1; S_1 \| b_2; S_2] \stackrel{IO}{=} \{p\}[(b_1; S_1)^2 \| (b_2; S_2)^2].$$

Inductively we obtain for all $j \in \mathbb{N}$ that

$$\{p\}([b_1; S_1 \| b_2; S_2])^j \stackrel{IO}{=} \{p\}[(b_1; S_1)^j \| (b_2; S_2)^j].$$

Applying CCL 4 once more, we finally obtain

$$\{p\}([b_1; S_1 \| b_2; S_2])^j; [\neg b_1 \| \neg b_2] \stackrel{IO}{=} \{p\}[(b_1; S_1)^j; \neg b_1 \| (b_2; S_2)^j; \neg b_2].$$

Summarizing:

$$
\begin{array}{rl}
& O_{cl}[\![\{p\}\textbf{while } b \textbf{ do } [S_1\|S_2] \textbf{ od}]\!] \\
= & \bigcup_{j\in\mathbb{N}} O_{cl}[\![\{p\}(b;[S_1\|S_2])^j;\neg b]\!] \\
\stackrel{(a)}{=} & \bigcup_{j\in\mathbb{N}} O_{cl}[\![\{p\}((b_1\wedge b_2);[S_1\|S_2])^j;(\neg b_1\wedge\neg b_2)]\!] \\
\stackrel{(b)}{=} & \bigcup_{j\in\mathbb{N}} O_{cl}[\![\{p\}([b_1\|b_2];[S_1\|S_2])^j;[\neg b_1\|\neg b_2]]\!] \\
\stackrel{(c)}{=} & \bigcup_{j\in\mathbb{N}} O_{cl}[\![\{p\}([b_1;S_2\|b_2;S_2])^j;[\neg b_1\|\neg b_2]]\!] \\
\stackrel{(d)}{=} & \bigcup_{j\in\mathbb{N}} O_{cl}[\![\{p\}[(b_1;S_1)^j;\neg b_1\|(b_2;S_2)^j;\neg b_2]]\!].
\end{array}
$$

Next we prove that, for all $j \in \mathbb{N}$,

$$
\begin{array}{rl}
& O_{cl}[\![\{p\}[(b_1;S_1)^j;\neg b_1\|(b_2;S_2)^j;\neg b_2]]\!]. \\
= & \bigcup_{k\in\mathbb{N}} O_{cl}[\![\{p\}[(b_1;S_1)^j;\neg b_1\|(b_2;S_2)^k;\neg b_2]]\!].
\end{array}
$$

We prove this claim by showing that for any $k \neq j$

$$O_{cl}[\![\{p\}[(b_1; S_1)^j; \neg b_1 \| (b_2; S_2)^k; \neg b_2]]\!] = \emptyset.$$

Assume that $k \neq j$, and without loss of generality, that $j < k$. By (d) we have that

$$\begin{array}{ll} & \{p\}[(b_1; S_1)^j; \neg b_1 \| (b_2; S_2)^k; \neg b_2] \\ \stackrel{IO}{=} & \{p\}([b_1; S_1 \| b_2; S_2])^j; [\neg b_1 \| b_2]; (S_2; b_2)^{k-j-1}; S_2; \neg b_2 \end{array}$$

By properties (i) and (ii) the loop invariant I holds at the state where the guard $\neg b_1 \wedge b_2$ is evaluated. However, by property (iii) this guard evaluates to "false", and cannot be passed. It follows that for $j \neq k$

$$O_{cl}[\![\{p\}[(b_1; S_1)^j; \neg b_1 \| (b_2; S_2)^k; \neg b_2]]\!] = \emptyset.$$

We deduce by the above equivalences, the compositionality of the reactive-event-sequence semantics $\mathcal{RA}$, and the fact that $\bigcup_{j\in\mathbb{N}}$ distributes over $\tilde{\|}$:

$$
\begin{aligned}
& O_{cl}[\![\{p\}\textbf{while } b \textbf{ do } [S_1\|S_2] \textbf{ od}]\!] \\
= \;& \textstyle\bigcup_{j\in\mathbb{N}} O_{cl}[\![\{p\}(b;[S_1\|S_2])^j;\neg b]\!] \\
= \;& \textstyle\bigcup_{j\in\mathbb{N}}\bigcup_{k\in\mathbb{N}} O_{cl}[\![\{p\}[(b_1;S_1)^j;\neg b_1\|(b_2;S_2)^k;\neg b_2]]\!] \\
= \;& \textstyle\bigcup_{k\in\mathbb{N}} O_{cl}[\![\{p\}[\textbf{while } b_1 \textbf{ do } S_1 \textbf{ od}\|(b_2;S_2)^k;\neg b_2]]\!] \\
= \;& O_{cl}[\![\{p\}[\textbf{while } b_1 \textbf{ do } S_1 \textbf{ od}\|\textbf{while } b_2 \textbf{ do } S_2 \textbf{ od}]]\!].
\end{aligned}
$$

Since

$$\{p\}\textbf{while } b \textbf{ do } [S_1 \| S_2] \textbf{ od} \tag{1}$$

$$\stackrel{IO}{=} \quad \{p\}[\textbf{while } b_1 \textbf{ do } S_1 \textbf{ od} \| \textbf{while } b_2 \textbf{ do } S_2 \textbf{ od}]. \tag{2}$$

both loops satisfy the same pre- and postconditions. Thus, the communication closedness of (2) follows from from communication closedness of (1).

Example: Set-Partitioning

Example: Set-Partitioning

- Given two disjoint, nonempty and finite sets of integers S_0 and T_0, e.g.

$$S_0 = \{3, 8, 9\} \text{ and } T_0 = \{1, 4\}.$$

- $S_0 \cup T_0$ must be partitioned into two subsets S and T such that
  - $|S_0| = |S|$,
  - $|T_0| = |T|$,
  - every element of S is smaller than any element of T, e.g.

  $S_0 \cup T_0 = \{3, 8, 9, 1, 4\} \longrightarrow S = \{1, 3, 4\}$ and $T = \{8, 9\}$.

- Algorithmic idea:

"Exchange max(S) with min(T) until maxi–mum of S is smaller than minimum of T."

Example: Set-Partitioning

Pre- and Postcondition for set-partitioning algorithm:

$$pre \overset{\text{def}}{=} \left\{ S = S_0 \neq \emptyset \wedge T = T_0 \neq \emptyset \wedge S \cap T = \emptyset \right\}.$$

$$post \overset{\text{def}}{=} \left\{ \begin{array}{c} |S| = |S_0| \wedge |T| = |T_0| \wedge S \cup T = S_0 \cup T_0 \\ \wedge max(S) < min(T) \end{array} \right\}.$$

Algorithm:

$\{pre\}$

$max, min := max(S), min(T);$

while $max > min$ **do**

  (* exchange max with min *);

  $max, min := max(s), min(T);$

od;

$\{post\}$

Example: Set-Partitioning

$\underline{P_{SV}}$:*(Proof outline: O&G system for closed programs)*

$\{pre\}$

$[max := max(S) \parallel min := min(T)];$

$\{q_0\}$**skip**;$\{I\}$

while $max > min$ **do**

$\{I \wedge max > min\}$**skip**;$\{q_1\}$

$[S := (S \backslash \{max\}) \cup \{min\} \parallel T := (T \backslash \{min\}) \cup \{max\}]$

$\{q_2\}$

$[max := max(S) \parallel min := min(T)];$

$\{I\}$

od;

$\{I \wedge max \leq min\}$**skip**;$\{post\}$

Assertions P_{SV}:

$$pre = \left\{ S = S_0 \neq \emptyset \wedge T = T_0 \neq \emptyset \wedge S \cup T = \emptyset \right\}.$$

$$q_0 \stackrel{\text{def}}{=} \left\{ \begin{array}{c} S = S_0 \neq \emptyset \wedge T = T_0 \neq \emptyset \wedge S \cup T = \emptyset \\ \wedge min = min(T) \wedge max = max(S) \end{array} \right\}.$$

$$q_1 \stackrel{\text{def}}{=} q_2[S, T \;/\; (S \backslash \{min\}) \cup \{max\}, (T \backslash \{max\}) \cup \{min\}].$$

$$q_2 \stackrel{\text{def}}{=} \left\{ \begin{array}{c} |S| = |S_0| \wedge |T| = |T_0| \wedge S \cup T = S_0 \cup T_0 \\ \wedge S \cap T = \emptyset \wedge S \neq \emptyset \wedge T \neq \emptyset \end{array} \right\}.$$

$$I \stackrel{\text{def}}{=} \left\{ \begin{array}{c} |S| = |S_0| \wedge |T| = |T_0| \wedge S \cup T = S_0 \cup T_0 \\ \wedge S \cap T = \emptyset \wedge S \neq \emptyset \wedge T \neq \emptyset \\ \wedge min = min(T) \wedge max = max(S) \end{array} \right\}.$$

$$post = \left\{ \begin{array}{c} |S| = |S_0| \wedge |T| = |T_0| \wedge S \cup T = S_0 \cup T_0 \\ \wedge max < min(T) \end{array} \right\}.$$

Example: Set-Partitioning

$\underline{P_{init}}$:*(Communication-based)*

$\{pre'\}$
$[max := max(S) \parallel min := min(T)];$
$\{q'_0\}$
$[[\textbf{send}(C, max) \parallel \textbf{receive}(D, mn)] \parallel [\textbf{receive}(C, mx) \parallel \textbf{send}(D, min)]]\,;$
$\{q'_{01}\}$ $\{I'\}$
while $max > min$ **do**
$\quad\{I' \wedge max > min\}$ $\{q'_1\}$
$\quad[S := (S \backslash \{max\}) \cup \{mn\} \parallel T := (T \backslash \{min\}) \cup \{mx\}];$
$\quad\{q'_2\}$
$\quad[max := max(S) \parallel min := min(T)];$
$\quad\{q'_{21}\}$
$\quad[[\textbf{send}(C, max) \parallel \textbf{receive}(D, mn)] \parallel [\textbf{receive}(C, mx) \parallel \textbf{send}(D, min)]]\,;$
$\quad\{I'\}$
od;
$\{I' \wedge max \leq min\}$ $\{post'\}$

Example: Set-Partitioning

Assertions P_{init}:

$$pre' \stackrel{\text{def}}{=} \left\{ \begin{array}{c} pre \\ \wedge \neg C.full \wedge C.sent = C.received \\ \wedge \neg D.full \wedge D.sent = D.received \end{array} \right\}.$$

$$q_0' \stackrel{\text{def}}{=} \left\{ \begin{array}{c} q_0 \\ \wedge \neg C.full \wedge C.sent = C.received \\ \wedge \neg D.full \wedge D.sent = D.received \end{array} \right\}.$$

$$q_{01}' \stackrel{\text{def}}{=} \left\{ \begin{array}{c} q_0' \\ \wedge mn = min \wedge mx = max \end{array} \right\}.$$

$$q_1' \stackrel{\text{def}}{=} \left\{ \begin{array}{c} q_1 \\ \wedge \neg C.full \wedge C.sent = C.received \\ \wedge \neg D.full \wedge D.sent = D.received \end{array} \right\}.$$

Example: Set-Partitioning

$$q'_2 \stackrel{\text{def}}{=} \left\{ \begin{array}{c} q_2 \\ \wedge\neg C.full \wedge C.sent = C.received \\ \wedge\neg D.full \wedge D.sent = D.received \end{array} \right\}.$$

$$q'_{21} \stackrel{\text{def}}{=} \left\{ \begin{array}{c} q'_2 \\ \wedge min = min(T) \wedge max = max(S) \end{array} \right\}.$$

$$I' \stackrel{\text{def}}{=} \left\{ \begin{array}{c} I \\ \wedge\neg C.full \wedge C.sent = C.received \\ \wedge\neg D.full \wedge D.sent = D.received \\ \wedge mn = min \wedge mx = max \end{array} \right\}.$$

$$post' \stackrel{\text{def}}{=} \left\{ \begin{array}{c} post \\ \wedge\neg C.full \wedge C.sent = C.received \\ \wedge\neg D.full \wedge D.sent = D.received \end{array} \right\}.$$

By CCL 3:

$$
\begin{array}{l}
[\ \ max := max(S) \parallel min := min(T)\ \ ];\\
\left[\begin{array}{ll} & [\mathbf{send}(C, max) \parallel \mathbf{receive}(D, mn)] \\ \parallel & [\mathbf{receive}(C, mx) \parallel \mathbf{send}(D, min)] \end{array}\right];
\end{array}
$$

$$\stackrel{\text{IO}}{=}$$

$$
\left[\begin{array}{l} max := max(S); \\ \left[\begin{array}{l} \mathbf{send}(C, max) \parallel \\ \mathbf{receive}(D, mn) \end{array}\right] \end{array} \middle\| \begin{array}{l} min := min(T); \\ \left[\begin{array}{l} \mathbf{send}(D, min) \parallel \\ \mathbf{receive}(C, mx) \end{array}\right] \end{array}\right];
$$

Example: Set-Partitioning

By CCL 4:

$$\left[\begin{array}{cccc} & S & := & (S\backslash\{max\})\cup\{min\} \\ \| & T & := & (T\backslash\{min\})\cup\{max\} \end{array}\right];$$

$$[max := max(S) \parallel min := min(T)];$$

$$\left[\begin{array}{ll} & [\mathbf{send}(C, max) \parallel \mathbf{receive}(D, mn)] \\ \| & [\mathbf{receive}(C, mx) \parallel \mathbf{send}(D, min)] \end{array}\right]$$

$$\stackrel{\text{IO}}{=}$$

$$\left[\begin{array}{l} S := (S\backslash\{max\})\cup\{min\}; \\ max := max(S); \\ \left[\begin{array}{l} \mathbf{send}(C, max) \parallel \\ \mathbf{receive}(D, mn) \end{array}\right] \end{array} \right\| \left. \begin{array}{l} T := (T\backslash\{min\})\cup\{max\}; \\ min := min(T); \\ \left[\begin{array}{l} \mathbf{send}(D, min) \parallel \\ \mathbf{receive}(D, mx) \end{array}\right] \end{array}\right]$$

So far we obtained

$\{pre'\}$

$$\left[\begin{array}{l} max := max(S); \\ \left[\begin{array}{l}\mathbf{send}(C, max) \parallel \\ \mathbf{receive}(D, mn)\end{array}\right]\end{array} \Bigg\| \begin{array}{l} min := min(T); \\ \left[\begin{array}{l}\mathbf{send}(D, min) \parallel \\ \mathbf{receive}(C, mx)\end{array}\right]\end{array}\right];$$

$\{q'_{01}\}\ \{I'\}$

while $max > min$ **do**

$\{I' \wedge max > min\}$

$$\left[\begin{array}{l} S := (S \backslash \{max\}) \cup \{min\}; \\ max := max(S); \\ \left[\begin{array}{l}\mathbf{send}(C, max) \parallel \\ \mathbf{receive}(D, mn)\end{array}\right]\end{array} \Bigg\| \begin{array}{l} T := (T \backslash \{min\}) \cup \{max\}; \\ min := min(T); \\ \left[\begin{array}{l}\mathbf{send}(D, min) \parallel \\ \mathbf{receive}(D, mx)\end{array}\right]\end{array}\right]$$

$\{I'\}$

od

$\{I' \wedge max \leq min\}\ \{post'\}$

Our next aim is to distribute the loop. Let

$$B_1 \stackrel{\text{def}}{=} \begin{cases} S := (S \backslash \{max\}) \cup \{min\}; \\ max := max(S); \\ [\textbf{send}(C, max) \| \textbf{receive}(D, mn)] \end{cases},$$

$$B_2 \stackrel{\text{def}}{=} \begin{cases} T := (T \backslash \{min\}) \cup \{max\}; \\ min := min(T); \\ [\textbf{send}(D, min) \| \textbf{receive}(C, mx)] \end{cases}.$$

By Theorem (Loop distribution):

$$\{pre'\}$$

$$\left[\begin{array}{l} max := max(S); \\ \left[\begin{array}{l} \mathbf{send}(C, max) \parallel \\ \mathbf{receive}(D, mn) \end{array}\right] \end{array} \middle\| \begin{array}{l} min := min(T); \\ \left[\begin{array}{l} \mathbf{send}(D, min) \parallel \\ \mathbf{receive}(C, mx) \end{array}\right] \end{array}\right];$$

$$\{q'_{01}\}$$

$$\left[\begin{array}{l} \mathbf{while}\ max > mn\ \mathbf{do} \\ \quad B_1 \\ \mathbf{od} \end{array} \middle\| \begin{array}{l} \mathbf{while}\ mx > min\ \mathbf{do} \\ \quad B_2 \\ \mathbf{od} \end{array}\right]$$

$$\{post'\}$$

Example: Set-Partitioning

We also obtained that

$$\{q'_{01}\}\ [\textbf{while}\ mx > min\ \textbf{do}\ B_1\ \textbf{od} \parallel \textbf{while}\ mx > min\ \textbf{do}\ B_2\ \textbf{od}]$$

is communication closed.

By CCL 4:

$\{pre'\}$

$$\left[\begin{array}{l} max := max(S); \\ \left[\begin{array}{l} \textbf{send}(C, max) \parallel \\ \textbf{receive}(D, mn) \end{array}\right]; \\ \textbf{while } max > mn \textbf{ do} \\ \quad S := (S \backslash \{max\}) \cup \{min\}; \\ \quad max := max(S); \\ \quad \left[\begin{array}{l} \textbf{send}(C, max) \parallel \\ \textbf{receive}(D, mn) \end{array}\right] \\ \textbf{od} \end{array}\right\| \begin{array}{l} min := min(T); \\ \left[\begin{array}{l} \textbf{send}(D, min) \parallel \\ \textbf{receive}(C, mx) \end{array}\right]; \\ \textbf{while } mx > min \textbf{ do} \\ \quad T := (T \backslash \{min\}) \cup \{max\}; \\ \quad min := min(T); \\ \quad \left[\begin{array}{l} \textbf{send}(D, min) \parallel \\ \textbf{receive}(D, mx) \end{array}\right] \\ \textbf{od} \end{array}\right]$$

$\{post'\}$

References

- W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods. Cambridge University Press (2001).
- W. Janssen, J. Zwiers. Specifying and proving communication closedness in Protocols. Technical Report 93-23, University of Twente (1993).
- Tz. Elrad, N. Francez, Decomposition of distributed programs into communication closed layers. In: Science of Computer Programming 2 (1982)